Applying domain and design knowledge to requirements engineering *

W. Lewis Johnson, Martin S. Feather

USC / Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292-6695

David R. Harris

Lockheed Sanders
MER24-1583, P.O. Box 2034
Nashua, NH 03061-2034

Abstract

This paper describes efforts to develop a transformation-based software environment which supports the acquisition and validation of software requirements specifications. These requirements may be stated informally at first, and then gradually formalized and elaborated. Support is provided for groups of requirements analysts working together, focusing on different analysis tasks and areas of concern. The environment assists in the validation of formalized requirements by translating them into natural language and graphical diagrams, and testing them against a running simulation of the system to be built. Requirements defined in terms of domain concepts are transformed into constraints on system components. The advantages of this approach are that specifications can be traced back to requirements and domain concepts, which in turn have been precisely defined.

1 Introduction

We have built a demonstration requirements/specification environment called ARIES[1] which requirements analysts may use in evaluating system requirements and codifying them in formal specifications. This work helps to address several roadblocks in providing knowledge-based automated assistance to the process of developing formal specifications. One of the principal roadblocks is that formal specification languages are difficult to use in requirements acquisition, particularly by people who are not experts in logic. ARIES provides tools for the gradual evolution of acquired requirements, expressed in hypertext and graphical diagrams, into formal specifications. The analysts invoke transformations to carry out this evolution; in general, support for rapid and coordinated evolution of requirements is a major concern. ARIES is particularly concerned with problems that arise in the development of specifications of large systems. Specification reuse is a major concern, so that large specifications do not have to be written from scratch. Mechanisms are provided for dealing with conflicts in requirements, especially those arising when groups of analysts work together. Validation techniques, including simulation, deduction, and abstraction, are provided, to cope with the problem that large specifications are difficult to understand and reason about.

ARIES is a product of the ongoing Knowledge-Based Software Assistant (KBSA) program. KBSA, as proposed in the 1983 report by the US Air Force's Rome Laboratories [10], was conceived as an integrated knowledge-based system to support all aspects of the software life cycle. The ARIES effort builds on the results of earlier efforts at USC / ISI and Lockheed Sanders. Requirements analysis was addressed in Lockheed Sanders's Knowledge-Based Requirements Assistant [11]. ISI developed the Knowledge-Based Specification Assistant [19, 15, 14] to support specification construction, validation, and evolution. In ARIES there is no attempt to separate requirements analysis activities from specification activities; rather, specifications are viewed as the outcome of the process of acquiring requirements and formalizing them.

2 ARIES's Contribution

ARIES contains tools for *acquisition*, *review*, *analysis*, and *evolution*. Acquisition facilities allow analysts to build up system descriptions gradually. Review and analysis tools allow analysts to check for consistency, correctness, and ambiguity, and gauge completeness; they also help make systems descriptions understandable to non-computer specialists. Evolution tools sup-

*An extended version of this paper appeared in *Journal of Systems Integration*, vol. 1, pp. 283-320.

[1] ARIES stands for Acquisition of Requirements and Incremental Evolution of Specifications.

port modification, tracing, and evolution of requirements descriptions.

2.1 Acquisition and review tools

The acquisition tools in ARIES aim to capture initial statements of requirements as simply and directly as possible. A structured text facility is employed for managing textual information found in relevant documents or in informal engineering notes. To the extent that such documents already exist and can be linked to subsequent formal specifications, ARIES can make a strong contribution to the correctness and traceability of the completed specification. Since natural language by itself is often awkward and ambiguous as a medium for stating requirements, other notations familiar to analysts are likewise supported: state transition diagrams, information flow diagrams, taxonomies, decomposition hierarchies, as well as formal specification languages. We are experimenting with domain-specific notations, to make it possible for non-computer specialists to describe requirements. Importantly, these notations are all mapped onto a common representation internal to ARIES.

The review process in ARIES applies many of the same tools as acquisition, but in reverse. Information entered into the system in one notation may be presented in a different notation; making it easier to check specifications for correctness and completeness of specifications. However, not all notations are used in a symmetric fashion—for example, ARIES is able to translate from its formal internal descriptions to English, but it cannot translate English into formal requirements statements.

2.2 Evolution

Evolution mechanisms are central to requirements analysis in ARIES: requirements statements are expected to evolve gradually over time. *Evolution transformations* are the principal mechanism for evolution in ARIES. They are operators that modify system descriptions in a controlled fashion, affecting some aspects of a requirements statement while retaining others unchanged. They also propagate changes throughout a system description. Significant effort has been invested in ARIES in identifying evolution steps (both meaning-preserving and non-meaning-preserving) that routinely occur in the specification development process, and automating them in the form of evolution transformations.

2.3 Analysis

Analysis capabilities help analysts check for inconsistencies in proposed requirements, and explore consequences. Three basic types of analysis capabilities are provided. First, a simulation facility translates descriptions of required behavior into executable simulations. By running the simulations, an analyst can determine whether the stated requirements really guarantee satisfactory behavior. Deduction mechanisms propagate information through the system description, both to complete it and to detect conflicts and inconsistencies. Abstraction mechanisms employ evolution transformations to extract simplified views of the system description. These abstracted views are typically easier to validate, either through simulation or by inspection.

2.4 Reuse tools

Requirements may be defined by specializing and adapting existing requirements in ARIES's knowledge base of common requirements; this makes it easier to define requirements quickly and accurately. *Folders* are used in ARIES to capture, separate, and relate bodies of requirements information. The analysts can control the extent to which folders share information, and gradually increase the sharing as inconsistencies are reconciled. ARIES places a heavy emphasis on codification and use of domain knowledge in requirements analysis. Although a number of researchers have identified domain modeling as a key concern (e.g., Greenspan [4]), it is given short shrift in typical practice. Requirements analysis is usually narrowly focused on describing the requirements for a single system. This is problematic if an organization is interested in introducing more than one computer system into an environment, or when the degree of computerization of an organization is expected to increase over time. We have been modeling particular domains within ARIES, and experimenting with using such knowledge in the engineering of requirements for multiple systems.

3 An Example of Use

To demonstrate the power of the ARIES approach, and its ability to handle large complex specification problems, we have devoted significant effort to a single domain, namely air traffic control (ATC). The requirements documents for the the Federal Aviation Administration's Advanced Automation Program [13]

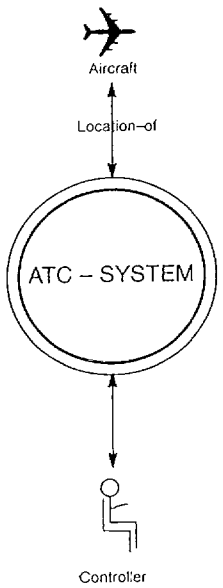

Figure 1: Initial context diagram of ATC system

were studied, and sections of these documents were formalized. Other documents concerning air traffic control procedures, as well as interviews with ATC experts, were also used as part of this study. The purpose of the study was to show that ARIES is potentially applicable to such problems, although we have not attempted to test it in use by software engineers on such projects.

The following discussion presents a particular thread of specification development, taken from the ATC domain. This example highlights the use of evolution transformations in specification development. Other capabilities will be illustrated throughout the paper.

Figure 1 shows an initial view of aircraft course monitoring. It is depicted here in a *context diagram*, a diagram showing the interactions between a system and its external environment, and the information that flows between them. In these diagrams, ovals denote processes, boxes and miscellaneous icons denote objects, and double circles indicate system boundaries.

The diagram distills course monitoring to its essential elements: the interaction between aircraft and the ATC system (ATC). This abstracted view of the ATC system is useful as a basis for stating course monitoring requirements. It is a natural abstraction for the domain, corresponding to the way flight procedures are commonly described in airmen's flight manuals [2]. We will not go into the specific details here of how far expected location and actual location are permitted to differ. Our concern is rather to ensure that course monitoring requirements stated from the airman's point of view can be transformed into specifications of system functionality, so that they can be integrated into the requirements specification. The transformed requirements should take into account the actual data interfaces of the proposed system.

Figure 2 shows a more detailed view of the ATC process. In this view, more of the agents of the proposed system are introduced, specifically radars and controllers. ATC is no longer viewed as a single agent; instead, there are two classes of agents, the air traffic control computer system and the controllers. The air traffic control system has as one of its subfunctions a process called Ensure-On-Course which examines the location of the aircraft, and compares it against the aircraft's expected location. If the two locations differ to a sufficient degree, ATC attempts to affect a course change, changing the location of the aircraft.

Now the system determines the locations of the aircraft as follows. The radar observes the aircraft and transmits a set of radar messages, indicating that targets have been observed at particular locations. A Track-Correlation function inputs these radar messages and processes them to produce a set of tracks. Each track corresponds to a specific aircraft; the locations associated with the tracks are updated when new radar messages are received. Meanwhile, expected aircraft locations are now computed from the aircraft flight plans, which in turn are input by the controllers. The Ensure-On-Course process is now modified so that it issues notifications to the controller (by signaling Must-Change-Course for an aircraft); the controller then issues commands to the aircraft over the radio.

In order to get to this more detailed level of description, a number of transformations must be performed. Most of the transformations have to do with defining the pattern of data flow through the system. We can suggest three options for obtaining this more detailed description.

First we can maintain a limited, very informal, link between the two descriptions — this in fact is what happens in most current practice. The detailed interconnections between abstractions are not stated explicitly and any attempt at traceability occurs at best through following a paper trail. It may be possible to say that description 2 follows description 1, but there is no record of the evolution (e.g., how was access to "location-of" data changed?) as we move toward a formal description.

Second, the interconnections can be manually derived and recorded — perhaps using a global replace command on a textual version of the stated informa-

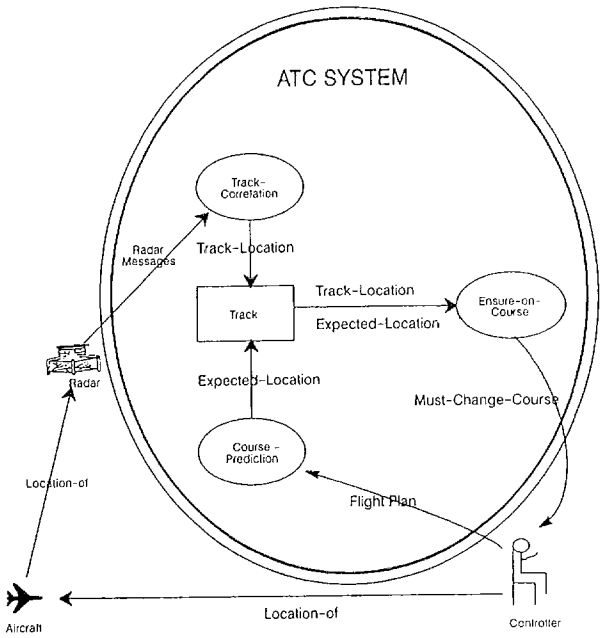

Figure 2: Detailed context diagram of the ATC system

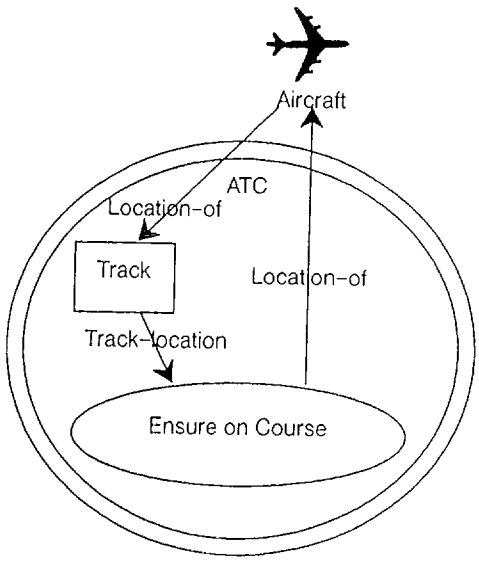

Figure 3: Intermediate context diagram of the ATC system

tion. Traceability is possible, but the process is tedious and error prone.

Third, evolution transformations can be employed to derive the detailed description from the simplified one. We have implemented a number of the evolution transformations required. The user must select the desired transformations, but it is ARIES's responsibility to check the transformations applicability conditions and ensure that all effects are properly handled.

The most important transformation in this example is called Splice-Data-Accesses. Figure 3 shows the result of ARIES application of this transformation to the version in Figure 1. It operates as follows. In the initial version Ensure-On-Course accesses aircraft locations directly. Splice-Data-Accesses is used to introduce a new class of object, called Track, which has a location that matches the aircraft's location. The Ensure-On-Course process is modified in a corresponding way to refer to the track locations instead of the aircraft locations.

This is a very typical example of how evolution transformations work. The transformation modifies one aspect of the specification (data flow) while keeping other aspects fixed (e.g., the functionality of Ensure-On-Course). It accomplishes this via systematic changes to the specification. In this case, the transformation scans the definition of Ensure-On-Course looking for references to Location-of; each of these is replaced with a reference to the Track-Location attribute of tracks.

Completing the derivation of this example requires further application of several transformations. Splice-Data-Accesses is applied again to introduce the object Radar-Message, which is an intermediate between Aircraft and Track. Maintain-Invariant-Reactively is invoked to construct processes for continuously updating the radar messages and the tracks. A transformation called Install-Protocol is used to introduce a notification protocol between the Ensure-On-Course process and the controller, so that Ensure-On-Course issues notifications to the controller whenever the location of the aircraft must be changed. A new process called Course-Prediction is added to compute expected locations from flight plans. Through this derivation the specification is gradually refined towards a version in which each system component interacts only with those data and agents that it will be able to interact with in the implemented system.

4 Mechanisms for Supporting Specification Evolution

In this section, we describe the major technical challenges we have undertaken to create ARIES.

4.1 Folders and workspaces

The primary units of organization for specifications and requirements knowledge are *workspaces* and *fold-*

ers. Each analyst interacting with ARIES has one or more private workspaces, which are collections of system descriptions that are to be interpreted in a common context. Whenever an analyst is working on a problem, it is in the context of a particular workspace of definitions appropriate for that problem. In order to populate a workspace, the analyst makes use of one or more folders, which are collections of reusable concept definitions. The ARIES knowledge base currently contains 122 folders comprising over 1500 concepts. Folders may contain either formal or informal descriptions. Reusable formal descriptions include precise definitions of reusable concepts; reusable informal descriptions include excerpts from published documents describing requirements of the domain, e.g., air traffic control manuals.

4.2 Reuse techniques

Folders are intended to deal with the problems of sharing and hiding information that arise in systems with large knowledge bases. They facilitate reuse of requirements descriptions. A number of other reuse techniques are being developed and explored: representation of multiple models, parameterized folders, specialization hierarchies, higher-order constraints, and transformations.

4.2.1 Representation of multiple models

Analysts may selectively incorporate models of concepts into their specifications, as in the following example. The ARIES knowledge base contains several alternative models for directions: as compass points (e.g., north, south, east, and west), as the number of degrees clockwise from magnetic north, or as multiples of ten degrees from magnetic north (used to mark the direction of runways). Each model is stored in a different folder, these folders are organized in a *folder specification* hierarchy.

These collections of models may be used as follows. During the initial stages of requirements of analysis, when the system description is not very detailed, an analyst can use the folder with the most general, and least detailed, model of the concept of interest. Then, as requirements are more detailed, and the analyst has a better understanding of what details need to be modeled, a model that is lower in the folder specialization hierarchy may be used. This incremental traversal of the folder specialization hierarchy helps ensure that the model that is ultimately selected captures the aspects of the concept being modeled that are needed, and does not include irrelevant detail.

4.2.2 Reuse through concept specialization

If an analyst needs to define a new concept, it is often advantageous to define it as a specialization of one or more abstract concepts, rather than construct a definition from scratch. Specialization hierarchies relate specific concepts to more general concepts, as in other knowledge-based systems. They also relate specific models of concepts to generic models of the same concepts. The ARIES knowledge base contains specialization hierarchies of types, relations, and events. Some important technical concerns had to be addressed in order to ensure that such specialization hierarchies are meaningful.

Consider, for example, two events, takeoff and move. Intuitively, it would make sense for takeoff to be a specialization of move: if an aircraft is taking off, it is also moving. However, the two events are likely to have different parameters. In the ARIES model, takeoff takes as input one parameter, the aircraft taking off. move, on the other hand, has three parameters: the object being moved, the agent doing the moving, and the location that the object will be moved to. A simple logical implication between the two concepts cannot be drawn, because the parameters of the two concepts do not match up.

The solution to this problem that we provide in ARIES is to *reify* the events and relations, i.e., to treat instances of them as objects. When an event starts, an object representing the event is created; when the event completes, the object is destroyed. Parameters of the events become attributes of the corresponding event objects. Subsumption for events and relations is then equivalent to type subsumption for the objects representing the events and relations. In the case of takeoff and move, subsumption is defined as follows. Move actions are represented as objects with three attributes: actor, actee, and destination. These are the names of the input parameters in the declaration of move. Takeoff actions are modeled as having four parameters and roles. One of them, ac, is an input parameter – the aircraft taking off. Another, destination, is an output parameter, bound to the aircraft's new location when the takeoff is completed. Two other roles, actee and actor, are bound to the value of the input parameter ac. (Note that these binding are specified in the roles attribute of takeoff.) When a takeoff event is initiated, a corresponding object is created, with attributes ac, actor, actee, and destination, corresponding to the parameters and roles. By the semantics of term subsumption, making takeoff a specialization of move means that every event object describing a takeoff must also be a well-formed move object.

4.2.3 Adaptation

The above techniques all enable analysts to construct new specifications by reusing portions of existing specifications and domain knowledge. However, it is unrealistic to expect all concepts to be used in a specification to be present in reusable form. Reuse techniques must be complemented with techniques for adapting and modifying existing knowledge. While informal reuse (i.e., cutting and pasting as in a text editor) is possible, we believe that there are many advantages to be gained by using evolution transformations to control this process. Evolution transformations make it possible to adapt concepts in the knowledge base in restricted, systematic ways, to avoid the introduction of errors during the adaptation process.

4.3 Acquisition and review

In order to make use of folders and reusable information, it is necessary to be able to view them, select from them, and add to them. These actions are done through an interface called the Presentation Facility, which makes ti possible to enter and view information through a variety of different notations.

All notations that ARIES supports are views of the same underlying system description representation. The notations, which we call "presentations," fall into the following categories.

- Graphical presentations are diagrams showing certain objects in the ARIES knowledge base, and the links interconnecting them. The graphical presentations in ARIES include specialization hierarchy diagrams, state transition diagrams, data or information flow diagrams, context diagrams such as those shown in Section 3, and functional decomposition diagrams.

- Spreadsheet presentations are tabular diagrams that allow analysts to enter requirements for a collection of components of the system description and interact with an underlying constraint propagation system which is maintaining dependency links among requirements statements.

- Formal presentations are detailed formal specification texts, e.g., those written in Reusable Gist.

- Natural language is used in initial acquisition to capture informal statements that will later be formalized. Machine-generated natural language is used for checking formal specifications against informal requirements, in requirements documents, and in explaining specifications to clients and others who are not experts in requirements modeling.

The key technical challenge in supporting multiple presentations has been to develop a common internal representation, the ARIES Metamodel, that will easily map to the notations of stereotypical views of systems (e.g., data flow arcs, system functional decomposition, state transitions, predicate calculus-like formalisms). Some metamodel concepts are relatively separable and easy to handle. For example, the type, relation, and event taxonomic diagrams are generated from the internal representation in an obvious way. Other concepts are highly interrelated. States are relations which are derived from a designated relation which has a parameter varying over a finite set of values.

The ARIES Presentation System is an architecture for defining interactive presentations linked to the ARIES Metamodel. It is implemented in CLX and CLUE, on top of X windows, and is operational on both the TI Explorer and the Sun. Each presentation definition includes a declarative description of the metamodel relations which are used to establish and link presentation pieces, and the editing and navigation actions (associated either with a presentation piece or the entire presentation). Editing actions match effect descriptions of transformations. Once the analyst edits a presentation, ARIES searches for and applies the evolution transformations which can make the required change.

4.4 Analysis and simulation

Analysis tools include a constraint propagation engine and an incremental static analyzer. Analysis tools are important in order to check for completeness and consistency. A constraint mechanism, derived from Steele's Constraint Language [23], has been incorporated into ARIES for general maintenance of constraints — bidirectional propagation, contradiction detection, retraction, and explanation. This mechanism is essential where there are interacting design properties (e.g., interplay between performance characteristics) and developers can use assistance in identifying when an interaction of requirements may not be achievable. An incremental static analyzer, a version of the static analyzer developed for the Specification Assistant [16], maintains calling and type information for the system description as it is being edited. It also does such things as detect specification freedoms which must be removed temporarily before simulation can be performed.

Simulation tools are useful in order to observe the behavior of a proposed system or its environment, in order to determine appropriate parameters for requirements or to discover unexpected or erroneous behavior. Simulation of vehicle behavior demonstrates, for example, how long it takes for traffic flow to return to normal after a light has changed, thus suggesting what the appropriate light duration should be based on the rate of traffic flow.

Simulations are constructed by means of a specially modified compiler which translates a subset of the ARIES Metamodel into Lisp and AP5, an in-core relational database [6]. Events described in the specification can compile either into ordinary Lisp functions, or into task objects to be scheduled by the simulator's task scheduler. Functional requirements in the form of invariants are compiled into rules which notify the analyst if and when they are violated [3].

Successful simulation analysis depends crucially upon the model of the system and environment chosen for simulation. When attempting to answer a specific validation question, it is useful to remove from consideration those features of the system which are not relevant to the question. Otherwise the simulation will generate volumes of useless information. Consider, for example, the question of whether a specification of a traffic signal permits the traffic lights to be red in all four directions. To answer this question, it may be convenient to ignore the distinction between green and amber, and just treat traffic lights as two-state devices, red and non-red. Furthermore, it may be useful at first to restrict analysis to the intersection of two one-way streets: if red lights are permitted in all directions in this case, they will also be permitted in the two-way street case. If a suitable abstraction can be found, validation can also be performed by inspection and constraint propagation.

Kevin Benner in our group is currently investigating which abstractions are most suitable for which kinds of analysis tasks. He is developing evolution transformations which construct the abstractions and designing the simulator to execute these abstractions. Together these form a powerful set of capabilities for specification validation.

4.5 Evolution transformations

As part of our earlier Specification Assistant work, we built a sizable library of evolution transformations, that is, transformations whose very purpose is to *change* the meaning of the specification to which they are applied. Like conventional correctness-preserving transformations, they blend computer power — the ability to conduct repeated, mechanical operations rapidly and reliably — and human intuition — knowing which transformation to apply when. They allow us to:

- build specifications incrementally,
- explain specifications incrementally, i.e., by going through the incremental record of their construction, and
- modify specifications by applying further evolutions.

In the ARIES project we are now addressing several deficiences in our earlier development. The focus of the Specification Assistant project was to make an initial exploration of this approach to specification construction and validation. Thus in populating our library of evolution transformations, we were motivated by the examples we studied (primarily those of a patient monitoring system, and a portion of an air-traffic control system). We built somewhat generalized versions of the evolution transformations necessary for these examples, but paid little attention to completeness or uniformity of our emerging library. We subdivided the library into categories of transformations (e.g., data-flow-modifying transformations, structure-adding transformations), but otherwise did little to support the user of the system in selecting the appropriate transformation. In ARIES we are addressing all of these deficiences.

4.5.1 Infrastructure to support evolution

A major goal of the ARIES project has been to support the user in selecting evolution transformations from a library, and in applying them. This library constitutes reusable knowledge about the *process* of requirements analysis, which complements the knowledge about the inputs and outputs of this process, i.e., knowledge of domains and systems. We now sketch the approach we are taking towards developing a usable evolution transformation library.

The representation of specification concepts enables efficient and effective modification of the semantic content of complex specifications. Having identified specification characteristics, we then chose a common representation for them, semantic nets — nodes connected by links, where the types of the nodes and links determine which characteristic they represent. For example, in the entity-relationship model, procedures and types will be represented by nodes; the type of a procedure's formal parameter is represented by linking

the node representing that procedure with the node representing that type. Changes to the specification induce the corresponding changes on these semantic net representations of the specification's characteristics. Each change can be expressed as a combination of creating and destroying nodes, and inserting and removing links between nodes. We have identified frequently recurring composites of these operations, for example, *splice* removes a direct link between two nodes, A and B say, and replaces it with two links via an intermediary, C say, so that A is linked to C and C is linked to B.

Finally, we characterized each evolution transformation in terms of the effects it induces on the semantic net representation of each of the above categories. Splice-Data-Accesses, illustrated in Section 3, is an example of a transformation that performs a splice along the information flow dimension. Likewise, an evolution transformation that introduces an intermediate specialization of some concept (e.g., given a specification containing type person and type airline-pilot, a specialization of person, we might introduce an intermediate type employed-person) is characterized as inducing a *splice* upon the specialization link structure. Similarly, an evolution transformation that wraps a statement inside a conditional is also characterized as inducing a *splice*, but upon the control-flow structure (the control flow link that led into the original statement now leads into the surrounding conditional statement, and there is a link from the conditional to the original statement).

These steps considerably improved the use and organization of our library of transformations in the following ways:

- Selection from the library — to select an evolution transformation, we give the characteristics of the changes we wish to induce on the specification, expressed as generic operations on the different characteristics of specifications. We distinguish between changes that we want to have happen, changes that we don't want to have happen, and changes that we don't care about.

- Coverage of the library — we can (crudely) estimate where our library lacks coverage by looking for useful combinations of generic changes on the different characteristics for which there are no evolution transformations that induce those changes.

- Uniformity of the library — seemingly unrelated evolution transformations that induce the same generic changes upon different characteristics can be seen to be similar, and are constructed to reflect this similarity.

In addition to augmenting the evolution transformations with generic descriptions of the effects they induce, we also augment them with explicit representations of their inputs (what they must be given), outputs (what new specification structure(s) they produce), and preconditions (what conditions must be true to guarantee that they will run correctly). These are represented in the same internal representation that ARIES uses for describing inputs, outputs, and constraints on events. Each aspect of the transformation may also be given a hypertext documentation string, as is customary for other concept definitions in the system. This makes it possible to employ the same presentation and explanation tools to transformations as are applicable to components of application system descriptions.

5 Related Work

The evolutionary approach to requirements specification has a number of precursors. Burstall and Goguen argued that complex specifications should be put together from simple ones, and developed their language CLEAR to provide a mathematical foundation for this construction process [5]. Goldman observed that natural language descriptions of complex tasks often incorporate an evolutionary vein – the final description can be viewed as an elaboration of some simpler description, itself the elaboration of a yet simpler description, etc., back to some description deemed sufficiently simple to be comprehended from a non-evolutionary description [9].

Fickas suggested the application of an AI problem-solving approach to specification construction [8]. Fundamental to his approach is the notion that the steps of the construction process can be viewed as the primitive operations of a more general problem-solving process, and are hence ultimately mechanizable. Continuing work in this direction is reported in [22] and [1].

The Requirements Apprentice, [20], developed as part of the Programmer's Apprentice project [21], addresses the early stages of the software development process, and includes similar techniques to those of the Programmer's Apprentice but operating on representations of requirements. Use of the Programmer's Apprentice is thus centered around selection of the appropriate fragment and its composition with the growing program, with application of minor transformations

to tailor these introduced fragments. In contrast, our approach has been centered around selection of the appropriate evolution transformations, and reformulating abstract descriptions of system behavior using such transformations. Yet in fact the two approaches are closely related. Many evolution transformations instantiate cliches as part of their function. We are currently exploring ways of making these cliches more explicit in our transformation system.

Karen Huff has developed a software process modeling and planning system that is in some ways similar to ours [12]. Her GRAPPLE language for defining planning operators influenced our representation of evolution transformations. Conversely, her meta-operators applying to process plans were influenced by our work on evolution transformations.

Kelly and Nonnenmann's WATSON system [17] constructs formal specifications of telephone system behavior from informal scenarios expressed in natural language. Their system formalizes the scenarios and then attempts to incrementally generalize the scenario in order to produce a finite-state machine. Their system is able to assume significant initiative in the formalization process, because the domain of interest, telephony, is highly constrained, and because the programs being specified, call control features, are relatively small. Our work is concerned with larger, less constrained design problems, where greater analyst involvement is needed.

The PRISMA project [18] is also a system for assisting in the construction of specifications from requirements. It supports multiple views of the emerging specification, where the views that they have explored are data-flow diagrams, entity relationship models, and petri nets. Each view is represented in the same underlying semantic-net formalism, yet represents a different aspect of the specification. This representation is suited to graphical presentation and admits to certain consistency and completeness heuristics whose semantics depend on the view being represented (e.g., the lack of an 'input' link in this representation in a data-flow diagram indicates a process lacking inputs; in an entity-relationship diagram it indicates an entity with no attributes; in a petri net diagram it indicates an event with no preconditions (prior events)). A paraphraser produces natural-language presentations of many of the kinds of information manipulated by the system (e.g., of the requirements information represented in the different views, of the agenda of tasks and advice for performing those tasks, and of the results of the heuristics that detect uses of requirements freedoms).

6 Summary

ARIES provides a variety of capabilities to support the process of requirements acquisition and analysis. These capabilities include acquisition, review, evolution support, analysis, and reuse support. These are intended to help analysts satisfy the conflicting goals of software requirements specification in a gradual and systematic way. The system as a whole focuses on the problems of describing systems from different viewpoints, and reconciling different viewpoints.

By building the ARIES prototype, we have been able to identify and offer solutions for many of the significant challenges which must be met in making knowledge-based requirements and specification development environments a reality. Specifically we have concentrated on supporting reuse of large domain independent and dependent knowledge-bases, providing multi-presentation acquisition along with significant automation support in the form of evolution transformations, specification analysis, simulations. We have developed mechanisms around many requirements support features including folders, reuse techniques, acquisition and review, analysis and simulation, evolution transformations, and traceability. Work on the project is ongoing; most of the capabilities envisioned for the system are already in place, but much work remains to be done.

7 Acknowledgements

The following people contributed to the work described here: Kevin Benner, Jay Myers, K. Narayanaswamy, Charles Rich, Jay Runkel, and Lorna Zorman. This work was sponsored Rome Laboratory contracts F30602-85-C-0221 and F30602-89-C-0103, and DARPA contract no. NCC-2-520. Views and conclusions contained in this paper are the authors' and should not be interpreted as representing the official opinion or policy of the U.S. Government or any agency thereof.

References

[1] J.S. Anderson and S. Fickas. A proposed perspective shift: Viewing specification design as a planning problem. In *Proceedings of the 5th International Workshop on Software Specification and Design, Pittsburgh, Pennsylvania*, pages 177–184. Computer Society Press of the IEEE, May 1989.

[2] ASA. *Airman's Information Manual.* Aviation Supplies and Academics, Seattle, WA, 1989.

[3] K. Benner. Using simulation techniques to analyze specifications. In *Proceedings of the 5th KBSA Conference*, pages 305–316, Syracuse, NY, 1990. Data Analysis Center for Software.

[4] A. Borgida, S. Greenspan, and J. Mylopoulos. Knowledge representation as the basis for requirements specifications. *IEEE Computer*, 18(4):82–91, 1985.

[5] R.M. Burstall and J. Goguen. Putting theories together to make specifications. In *Proceedings of the Fifth International Conference on Artificial Intelligence*, pages 1045–1058, August 1977.

[6] D. Cohen. *AP5 Manual.* USC-Information Sciences Institute, June 1989. Draft.

[7] A.M. Davis. *Software Requirements Analysis and Specification.* Prentice Hall, Englewood Cliffs, N.J., 1990.

[8] S. Fickas. A knowledge-based approach to specification acquisition and construction. Technical Report 86-1, CS Dept., University of Oregon, Eugene, 1986.

[9] N.M. Goldman. Three dimensions of design development. In *Proceedings, 3rd National Conference on Artificial Intelligence, Washington D.C.*, pages 130–133, August 1983.

[10] C. Green, D. Luckham, R. Balzer, T. Cheatham, and C. Rich. Report on a knowledge-based software assistant. In *Readings in Artificial Intelligence and Software Engineering.* Morgan Kaufmann, Los Altos, CA, 1986.

[11] D. Harris and A. Czuchry. The knowledge-based requirements assistant. *IEEE Expert*, 3(4), 1988.

[12] K.E. Huff and V.R. Lesser. The GRAPPLE plan formalism. Technical Report 87-08, U. Mass. Department of Computer and Information Science, April 1987.

[13] V. Hunt and A. Zellweger. The FAA's advanced automation system: Strategies for future air traffic control systems. *IEEE Computer*, 20(2):19–32, February 1987.

[14] W.L. Johnson. Deriving specifications from requirements. In *Proceedings of the 10th International Conference on Software Engineering*, pages 428–437, 1988.

[15] W.L. Johnson. Specification as formalizing and transforming domain knowledge. In *Proceedings of the AAAI Workshop on Automating Software Design*, pages 48–55, St. Paul, MN, 1988.

[16] W.L. Johnson and K. Yue. An integrated specification development framework. Technical Report RS-88-215, USC / Information Sciences Institute, 1988.

[17] V.E. Kelly and U. Nonnenmann. Reducing the complexity of formal specification acquisition. In *Proceedings of the AAAI-88 Workshop on Automating Software Design*, pages 66–72, St. Paul, MN, 1988.

[18] C. Niskier, T. Maibaum, and D. Schwabe. A look through PRISMA: Towards pluralistic knowledge-based environments for software specification acquisition. In *Proceedings, 5th International Workshop on Software Specification and Design, Pittsburgh, Pennsylvania, May*, pages 128–136. Computer Society Press of the IEEE, 1989.

[19] The KBSA Project. Knowledge-based specification assistant: Final report. unpublished, 1988.

[20] H.B. Reubenstein and R.C. Waters. The requirements apprentice: An initial scenario. In *Proc. of the 5th International Workshop on Software Specification and Design*, pages 211–218, Pittsburgh, PA, May 1989. Computer Society Press of the IEEE.

[21] C. Rich. *The Programmer's Apprentice.* ACM Press, Baltimore, MD, 1990.

[22] W.N. Robinson. Integrating multiple specificationss using domain goals. In *Proceedings, 5th International Workshop on Software Specification and Design, Pittsburgh, Pennsylvania, May*, pages 219–226. Computer Society Press of the IEEE, 1989.

[23] G.L. Jr. Steele. The definition and implementation of a computer programming language. Technical Report 595, MIT Artificial Intelligence Laboratory, 1980.